



DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Parcial de Organización del Computador I

Lic. en Ciencias de la Computación

22 de noviembre, 2020

Organización del Computador I

Estudiante	LU	Correo electrónico
Yulita, Federico	351/17	fyulita@dc.uba.ar



Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2610 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (++54 +11) 4576-3300

<https://exactas.uba.ar>

Introducción

Este parcial está dividido en cinco preguntas que se relacionan con la arquitectura RISC-V. Se trata de una especificación y no una implementación de un procesador concreto. Si bien la arquitectura es diferente a la que utilizamos en clase, encontrarán los mismos conceptos que estudiamos y trabajamos aplicados al diseño de esta ISA. Toda la información necesaria está disponible en la *Guía Práctica de RISC-V* que se puede acceder libremente en: <http://riscvbook.com/spanish/guia-practica-de-risc-v-1.0.5.pdf>. Les recomendamos que hagan primero una lectura completa de los primeros tres capítulos de la guía y luego intenten responder las preguntas. Los ejercicios varían en temática y complejidad así que también les recomendamos ordenar la resolución de los mismos como les resulte más sencillo.

Ejercicios

Ejercicio 1 Sabiendo que `a1 = 0xffffffff` ¿Cuánto queda almacenado en `a2` luego de realizar la operación: `andi a2, a1, 0xf00`?

Ejercicio 2 Explicar detalladamente el funcionamiento del siguiente programa en assembler:

```
main:
    addi a1, x0, 1
    addi a2, x0, 2
    bltu a2, a1, test
    sub a3, a1, a2
    jal end

test:
    sub a3, a2, a1

end:
    nop
```

Detallando el contenido de los registros modificados en cada instrucción (incluyendo el PC).

Ejercicio 3 ¿Qué significa que la arquitectura RISC-V sea modular? ¿Qué ventajas puede tener esto?

Ejercicio 4 ¿Cómo se resuelve la lógica de control (*branching*)? ¿Qué similitudes y/o diferencias existen con la máquina *Orga1*?

Ejercicio 5 ¿En qué consiste la *convención de llamadas*? ¿Qué registros se preservan? ¿Qué debe hacer la persona que escribe el código que sigue esta convención con los registros que no se preservan?

	x0	ra	sp	gp	tp	t0	t1	t2								
	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000								
	x0	s1	a0	a1	a2	a3	a4	a5								
PC	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000								
0x00000000	a6	a7	s2	s3	s4	s5	s6	s7								
	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000								
	s8	s9	s10	s11	t3	t4	t5	t6								
	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000								
	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+A	+B	+C	+D	+E	+F
0x00000000	0x00100593	-	-	-	0x00200613	-	-	-	0x00B66463	-	-	-	0x40C586B3	-	-	-
0x00000010	0x004000EF	-	-	-	0x40B606B3	-	-	-	0x00000013	-	-	-	-	-	-	-
	PC	IR	Instrucción				PC	Instrucción decodificada								
1	0x00000000	0x00100593	0000 0000 0001 0000 0000 0101 1001 0011	0x00000004				addi a1, x0, 1								
	Ejecución		a1 ← x0 + 1 = 0x00000000 + 0x00000001 = 0x00000001													
2	0x00000004	0x00200613	0000 0000 0010 0000 0000 0110 0001 0011	0x00000008				addi a2, x0, 2								
	Ejecución		a2 ← x0 + 2 = 0x00000000 + 0x00000002 = 0x00000002													
3	0x00000008	0x00B66463	0000 0000 1011 0110 0110 0110 0011 0011	0x0000000C				bltu a2, a1, test								
	Ejecución		if(a2 < a1): PC ← PC + 0x08													
4	0x0000000C	0x40C586B3	0100 0000 1100 0101 1000 0110 1011 0011	0x00000010				sub a3, a1, a2								
	Ejecución		a3 ← a1 - a2 = 0x00000001 - 0x00000002 = 0xFFFFFFF													
5	0x00000010	0x004000EF	0000 0000 0100 0000 0000 1110 1111 1111	0x00000014				jal end								
	Ejecución		ra ← PC = 0x00000014, PC ← PC + 0x00000004 = 0x00000018													
5	0x00000018	0x00000013	0000 0000 0000 0000 0000 0000 0001 0011	0x0000001C				nop								
	Ejecución		No operation													

Figura 1: Cartilla de seguimiento del programa del Ejercicio 2 para la máquina RISC-V.

Soluciones

Ejercicio 1 Esta operación hace un AND entre lo que esté en el registro `a1` y el inmediato `0xf00` extendido a 32 bits y lo almacena en el registro `a2`. Entonces:

```
a2 = a1 ⋅ sign_ext(0xf00)
    = 0xffffffff ⋅ 0xffffffff00
    = 0xffffffff00
```

El valor almacenado en `a2` es `0xffffffff00`.

Ejercicio 2

```
main:
    0x00: addi a1, x0, 1    (a1 ← x0 + sign_ext(0x01))
    0x04: addi a2, x0, 2    (a2 ← x0 + sign_ext(0x02))
    0x08: bltu a2, a1, test (if(a2 < a1): PC ← PC + sign_ext(0x08))
    0x0C: sub a3, a1, a2    (a3 ← a1 - a2)
    0x10: jal end          (ra ← PC, PC ← PC + sign_ext(0x04))

test:
    0x14: sub a3, a2, a1    (a3 ← a2 - a1)

end:
    0x18: nop              (halt)
```

Es importante remarcar que `x0 = 0x00000000` siempre, así que las primeras dos instrucciones le están asignando un valor a los registros llamados `a1` y `a2`. Este programa escrito en pseudo-C sería algo así:

```
int a1 = 0 + 1;
int a2 = 0 + 2;
if (a2 < a1) {
    // test
    int a3 = a2 - a1;
} else {
    int a3 = a1 - a2;
}
```

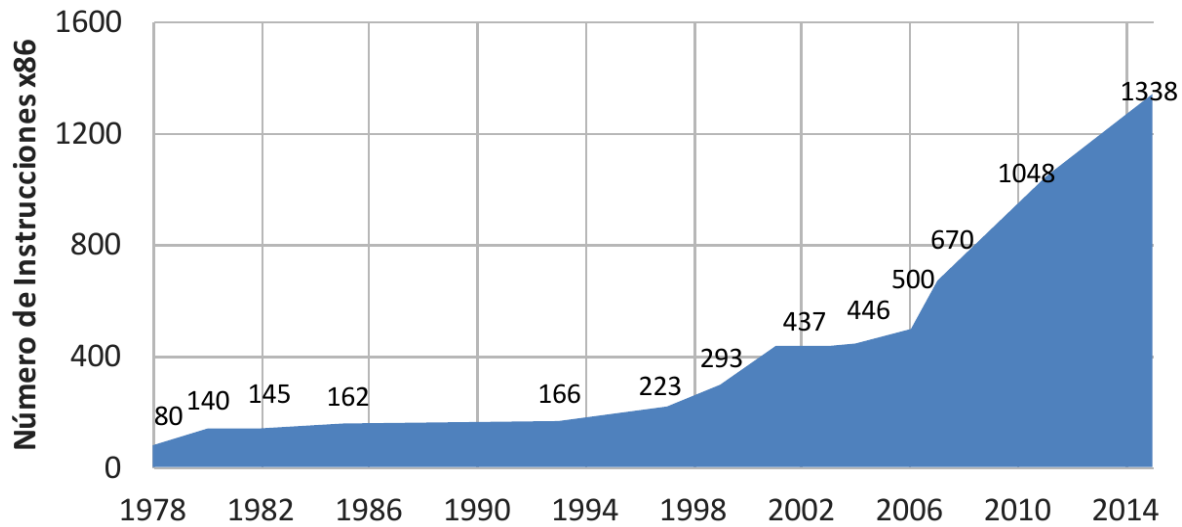


Figura 2: Crecimiento del conjunto de instrucciones x86 a lo largo de los años.

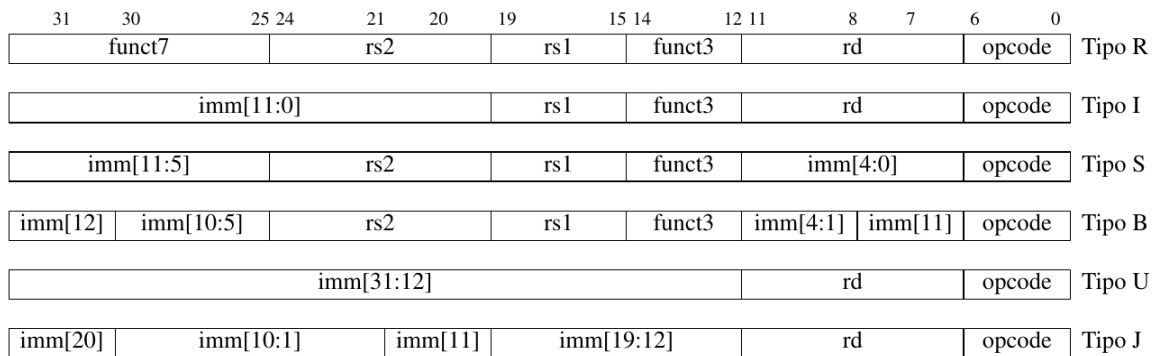


Figura 3: Formato de los distintos tipos de instrucciones RISC-V.

En este caso por los valores que se asignaron en los registros **a1** y **a2** está claro que la comparación **a2 < a1** resulta falsa y entonces el programa nunca pasa por la instrucción con la etiqueta **test** sino que asigna **a3 = -1** y termina yendo a la instrucción con la etiqueta **end**. En la **Figura 1** puede verse la cartilla de seguimiento de este programa.

Ejercicio 3 El autor compara las ISAs modulares con las incrementales. Las incrementales menciona que “los nuevos procesadores no solo implementan las nuevas extensiones, sino que además todas las instrucciones de ISAs anteriores”. Es decir, a medida que van saliendo nuevos procesadores y que se van agregando nuevas instrucciones y funcionalidades se van apilando, formando cada vez una ISA más extensa y compleja. En la **Figura 2** se encuentra un gráfico sacado del libro que muestra el crecimiento del conjunto de instrucciones x86 a lo largo de los años, desde su nacimiento en 1978 hasta el 2014. Esto es lo que significa una ISA incremental, la cantidad de instrucciones es una función creciente. En contraste, en una ISA modular el conjunto de instrucciones es fijo y a medida que se quiere agregar nueva funcionalidad a la ISA se hacen módulos que contienen estas nuevas instrucciones y que son opcionales. La ventaja de esto es que uno puede elegir qué módulos usar en cierto procesador en base a qué funcionalidad quiere que este cumpla y mantiene a la ISA relativamente reducida y simple. RISC-V cuenta con una ISA núcleo que es RV32I que funciona por sí misma y luego muchos módulos que extienden a la ISA y le agregan nueva funcionalidad: RV32M agrega multiplicación, RV32F agrega punto flotante de precisión simple, RV32D agrega punto flotante de precisión doble, etc.

Ejercicio 4 Existen seis tipos de instrucciones RISC-V: R para las instrucciones que usan dos registros de lectura y uno de escritura como **add**, **and** y **slt**; I para las que usan un registro de lectura, otro de escritura y un inmediato

Registro	Nombre ABI	Descripción	¿Preservado en llamadas?
x0	zero	Alambrado a cero	—
x1	ra	Dirección de retorno	No
x2	sp	Stack pointer	Sí
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Link register temporal/alternativo	No
x6–7	t1–2	Temporales	No
x8	s0/fp	Saved register/frame pointer	Sí
x9	s1	Saved register	Sí
x10–11	a0–1	Argumentos de función/valores de retorno	No
x12–17	a2–7	Argumentos de función	No
x18–27	s2–11	Saved registers	Sí
x28–31	t3–6	Temporales	No
f0–7	ft0–7	Temporales, FP	No
f8–9	fs0–1	Saved registers, FP	Sí
f10–11	fa0–1	Argumentos/valores de retorno, FP	No
f12–17	fa2–7	Argumentos, FP	No
f18–27	fs2–11	Saved registers, FP	Sí
f28–31	ft8–11	Temporales, FP	No

Figura 4: Tabla con la convención de nombres de los registros, la función que emplean y si se preservan en llamadas de funciones o no.

como `addi`, `andi` y `slli`; S para *stores* que usan dos registros de lectura y un inmediato como `sb`, `sh` y `sw`; B para *branches* que usan dos registros de lectura y un inmediato como `beq`, `bne` y `blt`; U para instrucciones que usan un registro de escritura y un inmediato como `lui` y `auipc`; y finalmente J para *jumps* que usan un registro de escritura y un inmediato como `jal`. En la **Figura 3** pueden verse los formatos de los distintos tipos de instrucciones RISC-V. Notemos que hay algunos formatos en los que a los inmediatos no se les puede cambiar el valor de `imm[0]` - es decir - el bit menos significativo está fijo. En particular, a las instrucciones de tipo B que son las instrucciones de branching esto es así y resulta que se asume que `imm[0] = 0` siempre. Esto es así porque las instrucciones son siempre múltiplos de dos bytes, entonces el bit menos significativo es siempre 0 porque las instrucciones deben ser números pares. RV32I puede comparar dos registros y saltar si el resultado es igual (`beq`), distinto (`bne`), mayor o igual (`bge`) o menor (`blt`). Además, también están las opciones sin signo, o *unsigned*, de las últimas dos operaciones (`bgeu` y `bltu` respectivamente). En caso que haya que hacer un salto entonces hay que multiplicar al valor inmediato de 12 bits por 2 (así le agregamos el bit menos significativo en 0), extendemos el signo a 32 bits y sumamos el resultado al PC.

Existen varias diferencias entre esta manera de manejar la lógica de control si la comparamos con Orga1. Orga1 lo que hace es hacer (o no) saltos en base al estado de los flags de la máquina. Por ejemplo, para la instrucción `JN` se fija si el flag Z está encendido y si es así hace el salto. Acá está la primera diferencia con RISC-V ya que esta última no utiliza flags que describan el resultado de la última operación que realizó la ALU. La siguiente diferencia es que en Orga1 cuando decide hacer el salto simplemente suma el inmediato al valor del PC, mientras que RISC-V tiene que multiplicar el inmediato por 2 y luego extender el signo. Finalmente, otra diferencia entre los saltos en Orga1 y en RISC-V es en la cantidad de instrucciones. Orga1 tiene 11 instrucciones para hacer saltos condicionales mientras que RISC-V tiene solo 6. El resto de los posibles saltos condicionales que no están explícitamente implementados como “mayor que” o “menor o igual” pueden hacerse simplemente intercambiando el orden de los operandos.

Ejercicio 5 Al llamar una función uno debe almacenar información en algún lugar, tanto para los argumentos como para los resultados. Acceder a información en registros es mucho más rápido que acceder a información en memoria así que es deseable almacenar la información que se requiera para la ejecución de una función en registros en vez de memoria. Como RISC-V tiene 32 registros entonces casi siempre hay suficiente espacio como para almacenar toda esta información en registros. Sin embargo, al tener tantos registros resulta conveniente establecer una convención de nombres y de funciones que van a distinguir cómo usamos estos registros.

Se van a distinguir registros en dos categorías: *temporary registers* y *saved registers*. Los temporary registers son registros que no garantizan conservar su valor a través de una llamada a función mientras que los saved registers

garantizan conservar su valor. En la **Figura 4** puede verse una tabla con la convención de llamadas descrita. Notemos que en total hay 16 temporary registers y 13 saved registers. El usuario que escriba código que siga esta convención de llamadas no debe escribir sobre los saved registers sino que sobre los temporary registers.